Unit Test White Paper

1. Purpose

This white paper describes unit testing methodologies and techniques that are commonly used in the industry. ***Refer to the CCS Standard for Unit Testing for guidelines specifically established for CCS, not this document!***

2. Applicability

This white paper applies to all new software units and adapted software units. This includes new software units being developed as well as any legacy software that is updated and/or converted and software developed for prototype purposes.

3. Reference Documents

- Jorgensen, Paul C., *Software Testing A Craftsman's Approach*, CRC Press 1995
- Bernard, Edward V., *Essays on Object-Oriented Software Engineering*, Prentice-Hall 1993
- McCabe & Associates, *Testing an Object Oriented Application*, Application Notes

4. Unit Test Objectives

The purpose of unit testing is to confirm that a unit is correctly coded and provides the capability allocated to it as defined in the design specification. Unit testing also ensures that error handling code is exercised, that any equations and algorithms are correctly implemented, and that there is no unacceptable loss of numerical precision or accuracy.

4.1 Unit Test Methods

There are several methods used to accomplish unit testing. This white paper will not go into all the details of how to implement each of the methods. The purpose of this white paper is to make the reader aware of what unit test methods are available and to guide the reader in making a decision of which methods will best achieve unit test objectives. There are reference documents cited in section 3 if a more detailed account of a particular unit test method is required.

A unit is defined as the smallest piece of software that can be independently tested (see section 4.3 on Object Oriented Issues). Unit testing consists of static testing and dynamic testing. Static testing is defined as program analysis without code execution. This involves visually examining source code and manually executing code to ensure that unit tests are satisfied. This type of testing is labor intensive and error- prone and should be done only for the simplest units where automated testing is impractical. Dynamic testing includes structural (white box) and functional (black box) testing. In structural testing, unit test cases are based on the internal control flow structure or data dependencies of the source code. Functional testing involves testing a unit against its external description without regard to its underlying implementation. The external description is contained in an external specification such as a design model or requirements specification and is used to develop unit test cases.
Unit testing should include structural testing. Structural testing involves using source code to construct a graphical representation of the code called a program graph. Using this program graph, structural testing techniques are employed. Unit testing should involve some amount of functional testing. Structural testing supplements functional testing for complex code or code which is mission critical.

4.1.1 Structural Testing

Structural testing begins with the generation of a program graph. Once a program graph is generated, source code is examined to determine which test method should be applied to best achieve unit test objectives. Each unit test method supports the generation of a suite of test cases.

4.1.1.1 Generating a Program Graph

A program graph is a pictorial presentation of source code. A program graph is a directed graph where program statements are represented by circles, and flow of control statements are represented by lines. The first step in generating a program graph is to identify programs statements and represent these statements on the graph using circles, otherwise called nodes. Each node is labeled with a number corresponding to the line number of the statement in the source code. (Identifying the nodes with line numbers may not be necessary if using an automated tool to generate program graphs. Most automated tools, not only generate the program graphs, but have a feature which displays the corresponding source code when a node is selected.) Nodes are then connected by lines, otherwise called edges. Edges represent the control flow of the code. Arrows are used to give the edges directionality. A program statement which contains a decision (e.g. if-then-else) is called a decision node. Decision nodes with control flow edges representing each decision option are included on the graph. In Figure 4.1.1.1 a simple program graph is shown with its corresponding source code. Upon generation of the program graph, test cases are generated with the intent to traverse every linearly independent path in the graph.

```
3.          int your_number;

4.          cout << “Enter a whole number: “;
5.          cin >> your_number;

6.          if (your_number % 2 == 0)
7.                      cout << “Your number is even \n”;
8.          else
                        cout << “Your number is odd. \n”;

9.           return 0;
```

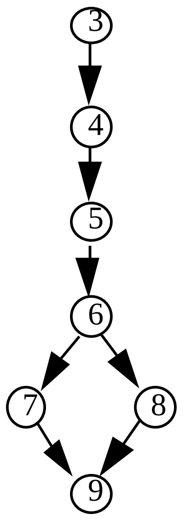

Figure 4.1.1.1-1 Example of a Program Graph

4.1.1.2 Structural Test Methods

It is not the intent of this white paper to suggest one testing method for all unit testing. Also, it is not intended that each method be applied to all code for unit testing. Certain test methods are more appropriate for testing certain units of code. The programmer should review the test methods and using the properties of the source code identify the appropriate structural testing method for a particular unit of code. The following paragraphs give brief explanations of various structural test methods. A table (4.1.2-1 Test Method Guidelines) is also provided which contains guidelines for determining which test method is best suited for unit code having certain characteristics. Three types of structural testing includes the Decision-to-Decision (DD) Path coverage method, the Basis Path coverage method, and the Decision-Use (DU) Path coverage method.

1) DD (Decision -to- Decision) Path - This method allows for the dissection of a unit of source code into separate paths, called DD paths, for testing. First a program graph is generated. Then the program graph is divided into several sequences of statements or paths. Each path begins with a decision statement and ends with the next decision statement in a unit of code. There are no internal branches (i.e. if-then-else statements) in the sequence of statements of each DD path. Test cases are then generated for each of the DD paths. There are several commercial tools on the market to aid the tester in generating DD paths. It has been observed and documented that when the DD path coverage method is properly used roughly 85% of all faults are revealed during unit test (reference Software Testing A Craftsman's Approach by Paul C. Jorgensen, CRC Press, Inc. 1995).

2) Basis Path (McCabe) - This method is based upon McCabes theory of cyclomatic complexity and graph theory. It begins with the generation of a program graph which is converted into a strongly connected graph by adding an edge from the sink node to the source node. To test all paths a "baseline" path is first determined, which should correspond to some "normal case" program execution. This is somewhat arbitrary but it is recommended to choice a path with as many decision nodes as possible. This baseline path is tested. Then the baseline path is retraced, in turn, for each decision path contained in the baseline path. Testing continues as each decision path is "flipped" (a decision is tested for "true" values then "false" values). The process is repeated until all options in each decision path have been tested.

   The cyclomatic number of a unit of code is calculated using the program graph and the algorithm $V(G) = e-n+p$ (where G represents the graph, e is the number of edges in G, n is the number of nodes, and p is the number of components). The cyclomatic number of a strongly connected graph is the number of linearly independent circuits in the graph (a circuit is a sequence of statements with no internal loops, but the initial node is the terminal node). The cyclomatic number of a program is equal to number of basis paths for a unit of code. Because greater cyclomatic complexity of a program yields greater numbers of tests to be generated and executed, it is a good practice to keep a program unit to a cyclomatic complexity of 10 or below.

3) DU (Decision Use) Path - The DU-Path method is structural testing with a focus on the points at which variables receive values and the points at which these values are used. This method starts with a program graph. From the program graph a table is generated identifying the nodes at which variables are introduced (non-executable statements such as const and variable declaration statements do not have to be considered defining nodes). The table includes the name of the variable, the node at which the variable is defined, and the node at which the variable is used. Using the table in conjunction with the program graph, the decision-use paths are defined and tested.

4.1.2 Functional Testing

Functional testing is based on a comparison of a units actual behavior with its specified behavior without regard to the code. It does not require the generation of program graphs. Focus is placed on testing functions, including inputs into a unit and the outputs expected as a result of running the code. Functional testing includes the following test methods: Boundary Value Analysis, Equivalency, and Decision Tables.

1) Boundary Value Analysis - This method focuses on test inputs at the lower and upper limits of their boundary space since this is where errors tend to occur. The basic idea is to generate test cases that use input values at their minimum value, just above the minimum value, at nominal value, just below their maximum value, and at their maximum value. If there are two or more inputs to a test the "single fault" rule is used. This rule is based on the assumption in reliability theory that states failures are rarely the result of the simultaneous occurrence of two (or more) faults. Therefore, only one

input at a time needs to vary from its minimum to maximum values. Boundary value analysis test cases are obtained by holding the values of all the inputs, except one, at their nominal values, and letting that one input vary. A function of n variables will yield 4n+1 test cases. If a function has two inputs, n_1 and n_2, the following set of test cases would apply:
(n_{1nom}- n_{2min}, n_{1nom}-n_{2min+}, n_{1nom}-n_{2max-}, n_{1nom}-n_{2max}, n_{1nom}-n_{2nom} ,n_{2nom}- n_{1min}, n_{1nom}-n_{1min+}, n_{1nom}-n_{1max-}, n_{1nom}-n_{1max}).

Boundary values for Boolean inputs are "true" and "false". The boundary value method works well with independent and physical inputs (i.e. dates are not independent so boundary testing may be meaningless, physical values include such variables as temperature, pressure, air speed, angle of instrument). This method is also good for systems that generate error messages.

2) Boundary Value Analysis Variations
There are many variations on the boundary value test method. The following methods are some variations for boundary value testing.

- robustness - This method is an extension of boundary value testing. Testing includes generating tests with inputs that have boundaries greater than maximum and less than minimum to test for exception handling. This testing is not only for inputs but also used to test for outputs that exceed expected values (e.g. the angle of an instrument is set greater than possible, thus an error message is expected with no turning of the instrument).
- worst case - This method rejects the "single fault" reliability theory. Testing involves allowing more than one test input to have an extreme value. This testing is very costly and should only be used when an error would result in an extremely costly failure.
- special value testing - This method of testing is based on ad hoc selection of input variables for testing, based on the experience of the tester as to the weak spots of the code.

3) Equivalency - This test method begins with partitioning a program into mutually disjoint subsets, then testing each subset with appropriate input values. For example, when testing code which calculates the dimensions of a triangle, testing with input values of 5,5,5 and 7,7,7 is not needed because these input values will all result in an isosceles triangle. Input values should be chosen which will result in outputs for equilateral, isosceles, scalene triangles. This method reduces redundant testing. Complete functional testing is ensured because if each partition is tested the "whole" is tested.

4) Decision Tables - This is the most rigorous of all functional test methods. It is ideal for situations where numerous combinations of actions are taken under varying sets of conditions (i.e. multiple decision paths). The first step is to identify all the possible conditions (inputs) and actions (outputs) for a function. Then construct a table with a column for each possible combination of inputs and outputs. Each column of the table defines a test case. Figure 4.1.2.-1 shows an example of a generic decision table.

CONDITION 1	yes	yes	no	no
CONDITION 2	yes	no	no	yes
ACTIONS	a	b	c	d

Figure 4.1.2-1Generic Decision Table

4.2 Test Efficiency and Effectiveness

Each tester needs to choose the "appropriate method" for unit testing. By using known attributes of the unit code, the most effective test methods can be selected. The test method should be well suited to the code and should avoid redundant testing. Well chosen hybrid testing that includes structural testing complimented by functional testing yields highly effective testing. Table 4.2-1 provides guidelines for choosing a test method.

TEST METHOD	TEST TYPE	GOOD UNDER THESE CONDITIONS
structural	DD-Path	good for all combinations, recommended with robust or equivalency testing
structural	Basis Path	good for all combinations, recommended with decision table testing
structural	DU-Path	in support of other path testing where program is computationally intensive and in control intensive programs where control variables are computed
functional	boundary	variables are physical quantities, variables are independent, single fault assumption
functional	equivalency	variables are physical quantities, variables are independent
functional	robustness	single fault assumption
functional	worst case	multiple -fault assumption
functional	decision tables	prominent if-then-else logic, logical relationships among input variables, input variables are dependent, multiple -fault assumption, significant exception handling, calculations involving subsets of the input variables, cause and effect relationships between inputs and outputs, high cyclomatic complexity (McCabe)

Table 4.2 -1 Test Method Guidelines

When using a hybrid test approach, the order in which testing is performed can be used to the testers advantage. The tester may perform functional testing first, followed by structural testing, or vice-versa. If functional testing is performed first, structural testing methods can be used as a cross check on functional test cases. For example, program graphs and structural path coverage methods can be used to reveal if functional tests traverse the same program path or if paths have not been traversed at all. Gaps and redundancies in testing can be identified and test cases can be developed to fill the gaps. On the other hand, if structural testing is performed first, then functional testing can be performed as a means to add test robustness to “suspect” code (code which is known to be critical as identified in the requirements or troublesome as identified as a result of the programmers experience with the code being develop). Boundary value and robustness testing can improve structural path testing. When additional coverage is needed to follow interesting paths special value testing can be performed.

4.2.1 Commercially Available Test Tools

1) <u>PureSoftware Purify Debugging Tool</u>

The Purify Tool supports code debugging by identifying run-time errors and memory leaks. Purify detects access errors and generates detailed diagnostic messages allowing tracking of the errors. The tool is incorporated into the development environment simply by adding a single word to existing makefiles and then relinking the code. Purify monitors each memory access made by the unit of code under test. This includes new code, legacy code, shared or dynamically linked libraries, third party libraries, or standard libraries.

2) McCabe Visual Testing Toolset

Automated structured testing using the McCabe Visual Testing Toolset (VTT) supports basis path testing. A completed unit of code is loaded into VTT. The VTT displays the test paths associated with the code and calculates the cyclomatic complexity number. The complexity number, tells the tester the number of test cases to develop to fully test the unit. Once the test cases are run, the VTT indicates graphically and textually what code has been executed and if any code remains untested. The VTT reports on basis path coverage, branch coverage, and statement coverage.

4.2.2 Support Test Tools

Initiating tests may involve the use of a driver. A driver can be a previously tested unit, a commercial test driver, or a test driver written by the programmer. For all three cases, the driver will call the unit to be tested and will initialize or pass a set of input parameters to the unit. The driver should provide a convenient method for varying the input parameters or input conditions for the unit, to support error and range testing.

Stubs should be written for the units called by the tested unit. A stub is a “dummy” unit that is used in lieu of the actual unit during unit test.

4.3 Object Oriented Issues

Unit testing of Object Oriented systems is very similar to testing systems not developed using object oriented methods. The difference lies in defining a “unit” to be tested.

In traditional “functional” approaches to software development the basic testable unit is the subprogram or subroutine. A subprograms in functional decomposition methodologies is a piece of code with a well-defined interface, that performed a single specific function. These were good candidates for static testing in design and black-box testing once the module of code was written.

In object oriented programming code is not written as several separate functional units or subprograms with distinct interfaces. Subprograms are bound (encapsulated) within a larger entity called a “class”. Furthermore, a program consists of a separate specification for its interface from its implementation (information hiding). Therefore testing a subprogram in isolation is meaningless. A good “rule of thumb” in object oriented systems is to define each class (and each instance of that class) as a unit to be tested individually. Testing should include all attributes and all operations defined for the class (or instance of a class). Test for classes and subclasses (including multiple inheritance where a subclass is allowed to inherit more than one parent class). Traditional techniques (path coverage, boundary values) can be used to perform unit test.

5. Unit Test Plan Documentation

The units programmer should develop a test plan which includes a set of test cases to be performed by the programmer to verify unit code. Test plans should provide test coverage for all decision paths resulting from the control logic expressed in the source code, and should verify that the software design and requirements are satisfied.

Test results should be kept in logs. This includes the number of tests performed and the number of faults found. These metrics may be used to determine test effectiveness over time.